COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Simon Winwood, Toby Murray, June Andronick, Gerwin Klein

a = b = c = . . .

Content

NICTA

- ➜ Intro & motivation, getting started with Isabelle
- ➜ Foundations & Principles
  - Lambda Calculus
  - Higher Order Logic, natural deduction
  - Term rewriting
- ➜ **Proof & Specification Techniques**
  - Inductively defined sets, rule induction
  - Datatypes, recursion, induction
  - **Calculational reasoning**
  - Hoare logic, proofs about programs
  - Locales, Presentation

Last time ...

- ➜ fun, function
- ➜ Well founded recursion

DEMO
MORE FUN

CALCULATIONAL REASONING

The Goal

$$
\begin{aligned}
x \cdot x^{-1} &= 1 \cdot (x \cdot x^{-1}) \\
\ldots &= 1 \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \\
\ldots &= 1
\end{aligned}
$$

The Goal

$$
\begin{aligned}
x \cdot x^{-1} &= 1 \cdot (x \cdot x^{-1}) \\
\ldots &= 1 \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \\
\ldots &= 1
\end{aligned}
$$

Can we do this in Isabelle?

The Goal

$$
\begin{aligned}
x \cdot x^{-1} &= 1 \cdot (x \cdot x^{-1}) \\
\ldots &= 1 \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \\
\ldots &= 1
\end{aligned}
$$

Can we do this in Isabelle?

➜ Simplifier: too eager

The Goal

$$
\begin{aligned}
x \cdot x^{-1} &= 1 \cdot (x \cdot x^{-1}) \\
\ldots &= 1 \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \\
\ldots &= 1
\end{aligned}
$$

Can we do this in Isabelle?

➜ Simplifier: too eager

➜ Manual: difficult in apply style

The Goal

$$
\begin{aligned}
x \cdot x^{-1} &= 1 \cdot (x \cdot x^{-1}) \\
\ldots &= 1 \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \\
\ldots &= (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \\
\ldots &= (x^{-1})^{-1} \cdot x^{-1} \\
\ldots &= 1
\end{aligned}
$$

Can we do this in Isabelle?

➜ Simplifier: too eager

➜ Manual: difficult in apply style

➜ Isar: with the methods we know, too verbose

Chains of equations

NICTA

The Problem

$$\begin{aligned} a &= b \\ \dots &= c \\ \dots &= d \end{aligned}$$

shows $a = d$ by transitivity of $=$

Chains of equations

The Problem

$$\begin{array}{lcl} a & = & b \\ \ldots & = & c \\ \ldots & = & d \end{array}$$

shows $a = d$ by transitivity of $=$

Each step usually nontrivial (requires own subproof)

Chains of equations

NICTA

The Problem

$$\begin{aligned} a &= b \\ \ldots &= c \\ \ldots &= d \end{aligned}$$

shows $a = d$ by transitivity of $=$

Each step usually nontrivial (requires own subproof)

Solution in Isar:

➜ Keywords **also** and **finally** to delimit steps

Chains of equations

The Problem

$$\begin{array}{lcl} a & = & b \\ \ldots & = & c \\ \ldots & = & d \end{array}$$

shows $a = d$ by transitivity of $=$

Each step usually nontrivial (requires own subproof)

Solution in Isar:

- Keywords **also** and **finally** to delimit steps
- **. . .**: predefined schematic term variable, refers to right hand side of last expression

Chains of equations

NICTA

The Problem

$$\begin{array}{lcl} a & = & b \\ \ldots & = & c \\ \ldots & = & d \end{array}$$

shows $a = d$ by transitivity of $=$

Each step usually nontrivial (requires own subproof)

Solution in Isar:

➜ Keywords **also** and **finally** to delimit steps

➜ **. . .**: predefined schematic term variable,
refers to right hand side of last expression

➜ Automatic use of transitivity rules to connect steps

also/finally

have "$t_0 = t_1$" [proof]

also

also/finally

have "$t_0 = t_1$" [proof]

also

calculation register

"$t_0 = t_1$"

also/finally

have "$t_0 = t_1$" [proof]

also

have "$\ldots = t_2$" [proof]

calculation register

"$t_0 = t_1$"

also/finally

have "$t_0 = t_1$" [proof]	calculation register
also	"$t_0 = t_1$"
have "$\ldots = t_2$" [proof]	
also	"$t_0 = t_2$"

also/finally

NICTA

have "$t_0 = t_1$" [proof]	calculation register
also	"$t_0 = t_1$"
have "$\ldots = t_2$" [proof]	
also	"$t_0 = t_2$"
$\vdots$	$\vdots$
also	"$t_0 = t_{n-1}$"

also/finally

have "$t_0 = t_1$" [proof]	calculation register
also	"$t_0 = t_1$"
have "$\ldots = t_2$" [proof]	
also	"$t_0 = t_2$"
$\vdots$	$\vdots$
also	"$t_0 = t_{n-1}$"
have "$\cdots = t_n$" [proof]	

also/finally

have "$t_0 = t_1$" [proof]	calculation register
also	"$t_0 = t_1$"
have "$\ldots = t_2$" [proof]	
also	"$t_0 = t_2$"
$\vdots$	$\vdots$
also	"$t_0 = t_{n-1}$"
have "$\cdots = t_n$" [proof]	
finally	$t_0 = t_n$

also/finally

have "$t_0 = t_1$" [proof]	calculation register
also	"$t_0 = t_1$"
have "$\ldots = t_2$" [proof]	
also	"$t_0 = t_2$"
$\vdots$	$\vdots$
also	"$t_0 = t_{n-1}$"
have "$\cdots = t_n$" [proof]	
finally	$t_0 = t_n$
show P	

— 'finally' pipes fact "$t_0 = t_n$" into the proof

More about also

➜ Works for all combinations of $=$, $\leq$ and $<$.

More about also

➜ Works for all combinations of $=$, $\leq$ and $<$.

➜ Uses all rules declared as `[trans]`.

More about also

- Works for all combinations of $=$, $\leq$ and $<$.
- Uses all rules declared as `[trans]`.
- To view all combinations in Proof General:

  Isabelle/Isar $\longrightarrow$ Show me $\longrightarrow$ Transitivity rules

Designing [trans] Rules

have = "$l_1 \odot r_1$" [proof]

also

have "$\ldots \odot r_2$" [proof]

also

Designing [trans] Rules

have = "$l_1 \odot r_1$" [proof]
also
have "$\ldots \odot r_2$" [proof]
also

Anatomy of a [trans] rule:

➜ Usual form: plain transitivity $[\![l_1 \odot r_1; r_1 \odot r_2]\!] \Longrightarrow l_1 \odot r_2$

Designing [trans] Rules

NICTA

have = "$l_1 \odot r_1$" [proof]
also
have "$\ldots \odot r_2$" [proof]
also

Anatomy of a [trans] rule:

- Usual form: plain transitivity $[\![l_1 \odot r_1; r_1 \odot r_2]\!] \Longrightarrow l_1 \odot r_2$
- More general form: $[\![P\ l_1\ r_1; Q\ r_1\ r_2; A]\!] \Longrightarrow C\ l_1\ r_2$

Examples:

Designing [trans] Rules

have = "$l_1 \odot r_1$" [proof]
also
have "$\ldots \odot r_2$" [proof]
also

Anatomy of a [trans] rule:

- ➜ Usual form: plain transitivity $[\![l_1 \odot r_1; r_1 \odot r_2]\!] \Longrightarrow l_1 \odot r_2$
- ➜ More general form: $[\![P\ l_1\ r_1; Q\ r_1\ r_2; A]\!] \Longrightarrow C\ l_1\ r_2$

Examples:

- ➜ pure transitivity: $[\![a = b; b = c]\!] \Longrightarrow a = c$

Designing [trans] Rules

have = "$l_1 \odot r_1$" [proof]

also

have "$\ldots \odot r_2$" [proof]

also

Anatomy of a [trans] rule:

- Usual form: plain transitivity $\llbracket l_1 \odot r_1; r_1 \odot r_2 \rrbracket \Longrightarrow l_1 \odot r_2$
- More general form: $\llbracket P\ l_1\ r_1; Q\ r_1\ r_2; A \rrbracket \Longrightarrow C\ l_1\ r_2$

Examples:

- pure transitivity: $\llbracket a = b; b = c \rrbracket \Longrightarrow a = c$
- mixed: $\llbracket a \leq b; b < c \rrbracket \Longrightarrow a < c$

Designing [trans] Rules

have = "$l_1 \odot r_1$" [proof]
also
have "$\ldots \odot r_2$" [proof]
also

Anatomy of a [trans] rule:

- ➜ Usual form: plain transitivity $[\![l_1 \odot r_1; r_1 \odot r_2]\!] \Longrightarrow l_1 \odot r_2$
- ➜ More general form: $[\![P\ l_1\ r_1; Q\ r_1\ r_2; A]\!] \Longrightarrow C\ l_1\ r_2$

Examples:

- ➜ pure transitivity: $[\![a = b; b = c]\!] \Longrightarrow a = c$
- ➜ mixed: $[\![a \leq b; b < c]\!] \Longrightarrow a < c$
- ➜ substitution: $[\![P\ a; a = b]\!] \Longrightarrow P\ b$

Designing [trans] Rules

have = "$l_1 \odot r_1$" [proof]
also
have "$\ldots \odot r_2$" [proof]
also

Anatomy of a [trans] rule:

- Usual form: plain transitivity $[\![l_1 \odot r_1; r_1 \odot r_2]\!] \Longrightarrow l_1 \odot r_2$
- More general form: $[\![P\ l_1\ r_1; Q\ r_1\ r_2; A]\!] \Longrightarrow C\ l_1\ r_2$

Examples:

- pure transitivity: $[\![a = b; b = c]\!] \Longrightarrow a = c$
- mixed: $[\![a \le b; b < c]\!] \Longrightarrow a < c$
- substitution: $[\![P\ a; a = b]\!] \Longrightarrow P\ b$
- antisymmetry: $[\![a < b; b < a]\!] \Longrightarrow P$

Designing [trans] Rules

NICTA

have = "$l_1 \odot r_1$" [proof]
also
have "$\ldots \odot r_2$" [proof]
also

Anatomy of a [trans] rule:

- ➜ Usual form: plain transitivity $[\![l_1 \odot r_1; r_1 \odot r_2]\!] \Longrightarrow l_1 \odot r_2$
- ➜ More general form: $[\![P\ l_1\ r_1; Q\ r_1\ r_2; A]\!] \Longrightarrow C\ l_1\ r_2$

Examples:

- ➜ pure transitivity: $[\![a = b; b = c]\!] \Longrightarrow a = c$
- ➜ mixed: $[\![a \leq b; b < c]\!] \Longrightarrow a < c$
- ➜ substitution: $[\![P\ a; a = b]\!] \Longrightarrow P\ b$
- ➜ antisymmetry: $[\![a < b; b < a]\!] \Longrightarrow P$
- ➜ monotonicity: $[\![a = f\ b; b < c; \bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y]\!] \Longrightarrow a < f\ c$

DEMO

HOL as programming language

NICTA

We have

- ➜ numbers, arithmetic
- ➜ recursive datatypes
- ➜ constant definitions, recursive functions

HOL as programming language

We have

- ➜ numbers, arithmetic
- ➜ recursive datatypes
- ➜ constant definitions, recursive functions
- ➜ = a functional programming language
- ➜ can be used to get fully verified programs

Executed using the simplifier.

HOL as programming language

We have

- ➜ numbers, arithmetic
- ➜ recursive datatypes
- ➜ constant definitions, recursive functions
- ➜ = a functional programming language
- ➜ can be used to get fully verified programs

Executed using the simplifier. But:

- ➜ slow, heavy-weight
- ➜ does not run stand-alone (without Isabelle)

Generating ML code

Generate stand-alone ML code for

- ➜ datatypes
- ➜ function definitions
- ➜ inductive definitions (sets)

Generating ML code

Generate stand-alone ML code for

- ➜ datatypes
- ➜ function definitions
- ➜ inductive definitions (sets)

Syntax (simplified):

code_module <structure-name> [**file** <name>]

contains

<ML-name> = <term>
⋮
<ML-name> = <term>

Generates ML stucture, puts it in own file or includes in current context

Value and Quickcheck

Evaluate big terms quickly:

value ”<term>”

- ➜ generates ML code
- ➜ runs ML
- ➜ converts back into Isabelle term

Value and Quickcheck

Evaluate big terms quickly:

value ”<term>”

- ➜ generates ML code
- ➜ runs ML
- ➜ converts back into Isabelle term

Try some values on current proof state:

quickcheck

- ➜ generates ML code
- ➜ runs ML on random values for numbers and datatypes
- ➜ increasing size of data set until limit reached

Customisation

➜ lemma instead of definition: **[code]** attribute

lemma [code]: ”(0 $<$ Suc n) = True” by simp

Customisation

➜ lemma instead of definition: **[code]** attribute

lemma [code]: "(0 $<$ Suc n) = True" by simp

➜ provide own code for types: **types_code**

types_code "$\times$" ("(_ */ _)")

Customisation

- lemma instead of definition: **[code]** attribute

  lemma [code]: "(0 $<$ Suc n) = True" by simp

- provide own code for types: **types_code**

  types_code "$\times$" ("(_ */ _)")

- provide own code for consts: **consts_code**

  consts_code "Pair" ("(_,/ _)")

Customisation

➜ lemma instead of definition: **[code]** attribute

lemma [code]: ”(0 < Suc n) = True” by simp

➜ provide own code for types: **types_code**

types_code ”×” (”(_ */ _)”)

➜ provide own code for consts: **consts_code**

consts_code ”Pair” (”(_,/ _)”)

➜ complex code template: patterns + **attach**

consts_code ”wfrec” (”\ <module>wfrec?”)
attach {* fun wfrec f x = f (wfrec f) x; *}

Code for inductive definitions

Inductive definitions are Horn clauses:

$$(0, \text{Suc } n) \in L$$

$$(n,m) \in L \Longrightarrow (\text{Suc } n, \text{Suc } m) \in L$$

Code for inductive definitions

Inductive definitions are Horn clauses:

$$(0, \text{Suc } n) \in L$$

$$(n,m) \in L \Longrightarrow (\text{Suc } n, \text{Suc } m) \in L$$

Can be evaluated like Prolog

Code for inductive definitions

Inductive definitions are Horn clauses:

$$(0, \text{Suc } n) \in L$$

$$(n,m) \in L \Longrightarrow (\text{Suc } n, \text{Suc } m) \in L$$

Can be evaluated like Prolog

code_module T

contains x = "λx y. (x, y) $\in$ L"

y = "(_, 5) $\in$ L"

generates

- ➜ something of type bool for x
- ➜ a possibly infinite sequence for y, enumerating all suitable _ in (_, 5) $\in$ L

DEMO

We have seen today ...

NICTA

- More fun
- Calculations: also/finally
- [trans]-rules
- Code generation